

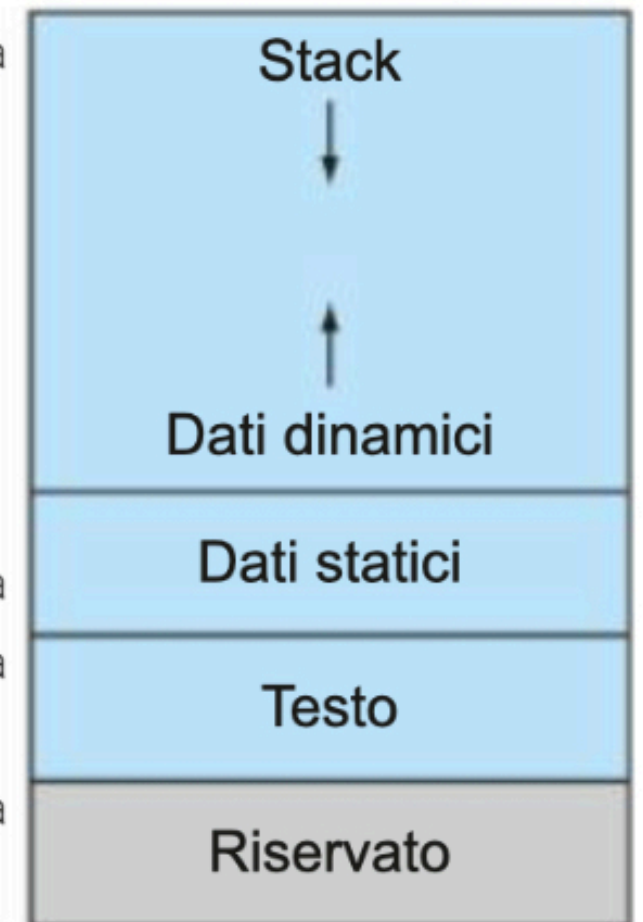
# Layout della memoria

- Testo: codice del programma
- Dati statici: variabili globali
  - Per esempio, variabili statiche in C, array costanti e stringhe
  - \$gp inizializzato all'indirizzo che permette  $\pm$ offset in questo segmento
- Dati dinamici: heap
  - Per esempio, malloc in C, new in C++/Java
  - Stack: spazio automatico

\$sp → 7fff fffc<sub>esa</sub>

\$gp → 1000 8000<sub>esa</sub>  
1000 0000<sub>esa</sub>

pc → 0040 0000<sub>esa</sub>  
0



# Dati di tipo carattere

- Insiemi di caratteri codificati su byte
  - ASCII: 128 caratteri
    - 95 grafici, 33 di controllo
  - Latin-1: 256 caratteri
    - ASCII + 96 caratteri grafici aggiuntivi
- Unicode: insieme di caratteri su 32 bit
  - Usati in Java, C++ (wide characters), ...
  - La maggior parte degli alfabeti del mondo + simboli
  - UTF-8, UTF-16: codifiche a lunghezza variabile

# Operazioni su byte/halfword

- Si potrebbero usare operazioni bit a bit

- MIPS load/store su byte/halfword

- Elaborazione di stringhe è un caso comune

`lb rt, offset(rs)`      `lh rt, offset(rs)`

- Estensione di segno su 32 bit in rt

`lbu rt, offset(rs)`      `lhu rt, offset(rs)`

- Estensione di zero su 32 bit in rt

`sb rt, offset(rs)`      `sh rt, offset(rs)`

- Semplicemente si memorizza il byte/halfword più a destra

# Esempio di copia di stringa

- Codice C (banale):
  - Stringa termina con null

```
void copia_stringa(char x[], char[] y)
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

- Indirizzi di x, y copiati in \$a0, \$a1
- i copiato in \$s0

# Esempio di copia di stringa

- Codice MIPS:

```
copia_stringa:
```

```
    addi $sp, $sp, -4      # prerarare lo stack per 1 elemento
```

```
    sw    $s0, 0($sp)     # salvare $s0
```

```
    add   $s0, $zero, $zero # i = 0
```

```
L1: add   $t1, $s0, $a1     # indirizzo di y[i] in $t1
```

```
    lbu   $t2, 0($t1)      # $t2 = y[i]
```

```
    add   $t3, $s0, $a0     # indirizzo di x[i] in $t3
```

```
    sb    $t2, 0($t3)      # x[i] = y[i]
```

```
    beq   $t2, $zero, L2    # uscire dal ciclo se y[i] == 0
```

```
    addi  $s0, $s0, 1       # i = i + 1
```

```
    j     L1               # prossima iterazione del ciclo
```

```
L2: lw    $s0, 0($sp)      # ripristinare $s0 salvato
```

```
    addi  $sp, $sp, 4       # estrarre 1 elemento dallo stack
```

```
    jr    $ra              # e ritornare
```

# Costanti a 32 bit

- La maggior parte delle costanti sono piccole
  - Il campo immediate a 16 bit è sufficiente
- Per le rare costanti a 32 bit

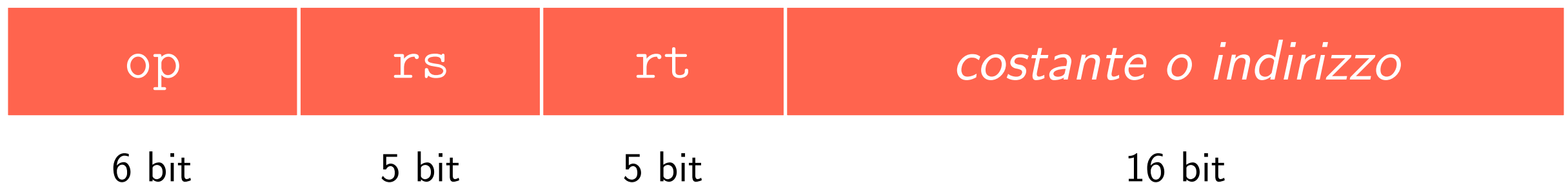
lui rt, constant

- Copia la costante a 16 bit nei 16 bit a sinistra di rt
- Resetta i 16 bit a destra di rt a 0

```
61 0000 0000 0000 0000 0000 0000 0111 1101
lui $s0, 61      0000 0000 0111 1101 0000 0000 0000 0000
2034 0000 0000 0000 0000 0000 1001 0000 0000
ori $s0, $s0, 2304 0000 0000 0111 1101 0000 1001 0000 0000
```

# Indirizzamento nei salti

- Le istruzioni di salto specificano
  - il codice dell'operazione, due registri, l'indirizzo destinazione
- La maggior parte dei salti indirizzano istruzioni vicini
  - In avanti o all'indietro



- Indirizzamento relativo a PC
  - Indirizzo destinazione =  $PC + \text{offset} \times 4$
  - PC è già incrementato di 4 prima di questo punto (!!!)

# Indirizzamento nei salti incondizionati

- Le destinazioni dei salti incondizionato (j e jal) potrebbero essere ovunque nel segmento text
- Si codifica l'indirizzo completo nell'istruzione (formato J)



- Indirizzamento del salto (pseudo)-diretto:
  - Indirizzo destinazione =  $PC_{31...28} : (\text{indirizzo} \times 4)$



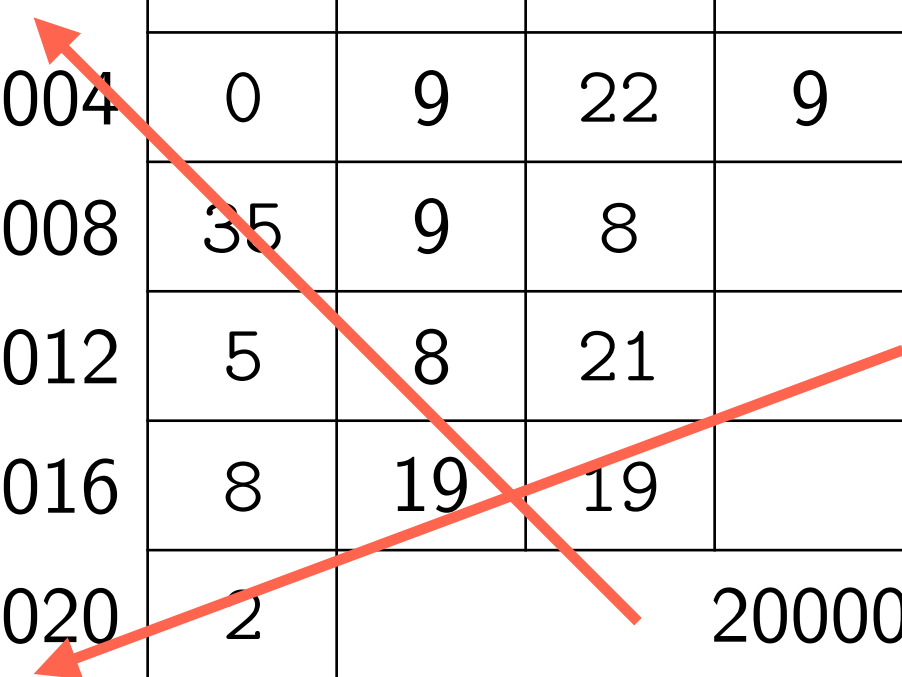
# Esempio di indirizzamento

- Codice del ciclo da un esempio precedente
- Si assuma che Loop sia alla locazione 80000

```
Loop: sll    $t1, $s3, 2
      add    $t1, $t1, $s6
      lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      addi   $s3, $s3, 1
      j      Loop
```

```
Exit: ...
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						



# Salti lontani

- Se la destinazione del salto è molto lontana per codificarla con un offset di 16 bit, l'assembler riscrive il codice
- Esempio:

```
beq $s0, $s1, L1
```



```
bne $s0, $s1, L2
```

```
j L1
```

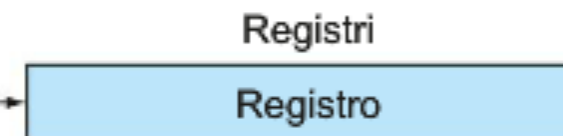
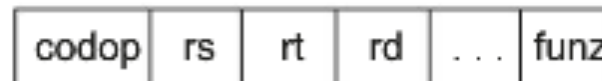
```
L2:
```

# Modalità di indirizzamento

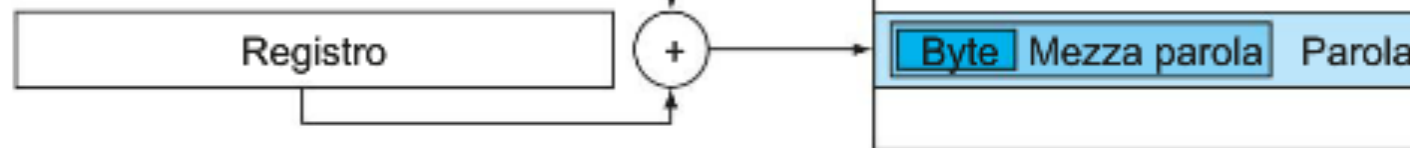
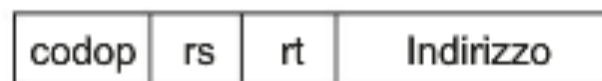
## 1. Indirizzamento immediato



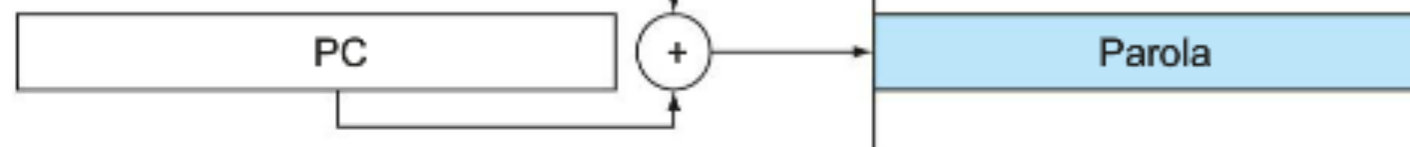
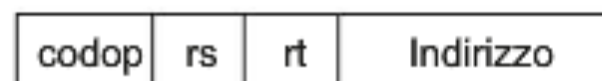
## 2. Indirizzamento tramite registro



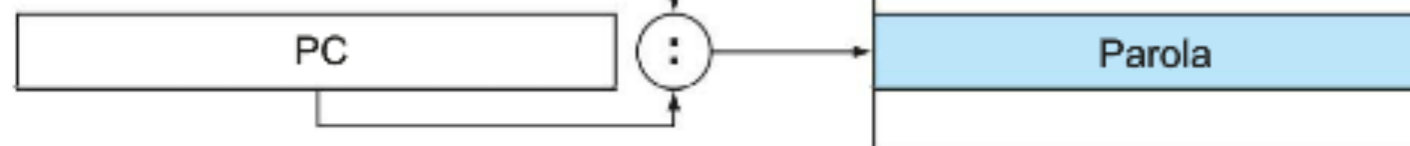
## 3. Indirizzamento tramite base



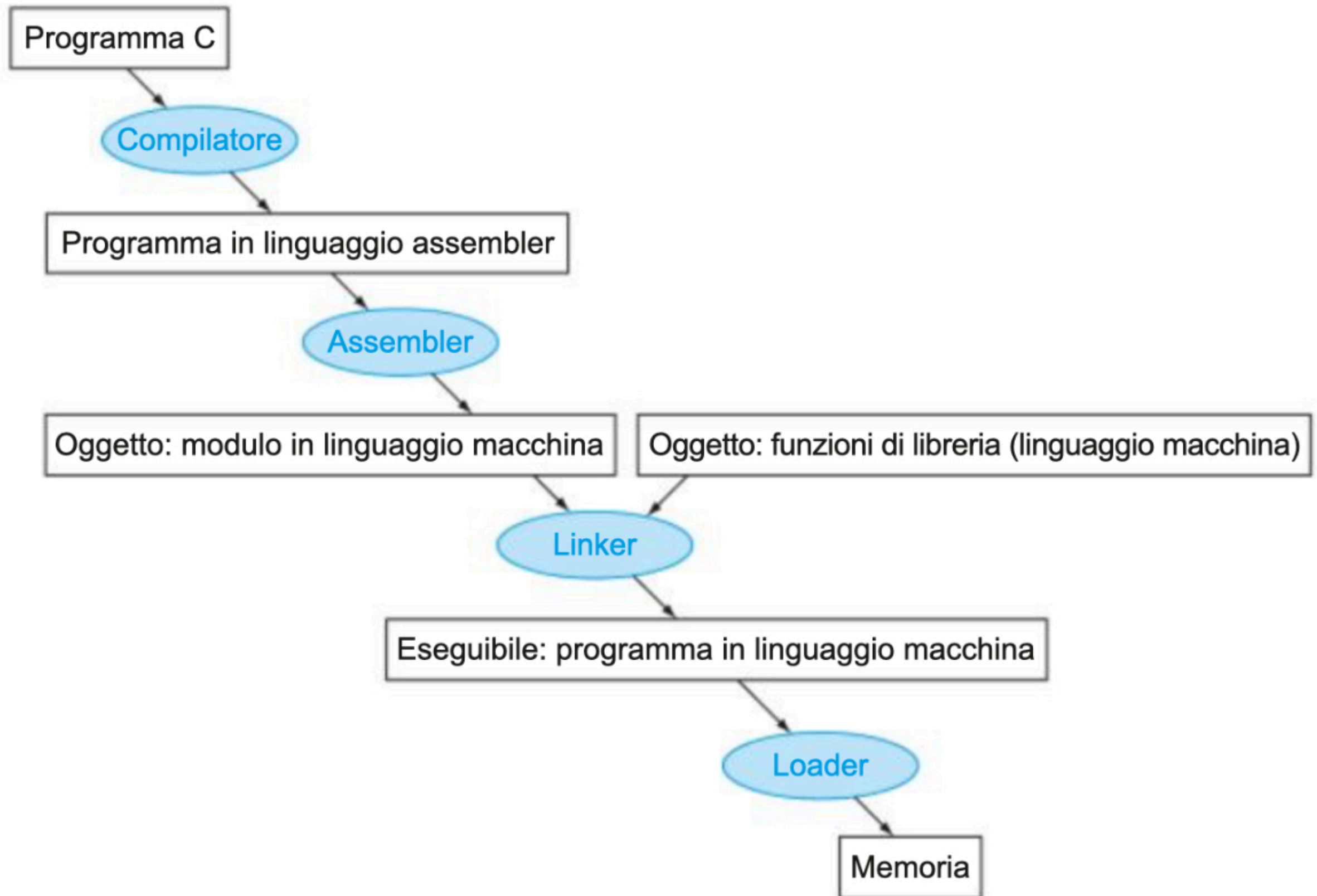
## 4. Indirizzamento relativo al PC



## 5. Indirizzamento pseudodiretto



# Traduzione e avvio



# Pseudo-istruzioni Assembler

- La maggior parte delle istruzioni assembler rappresentano uno a uno delle istruzioni macchina
- Pseudo-istruzioni: finzioni dell'immaginazione dell'assembler:

<code>move \$t0, \$t1</code>	<code>→</code>	<code>add \$t0, \$zero, \$t1</code>
<code>blt \$t0, \$t1, L</code>	<code>→</code>	<code>slt \$at, \$t0, \$t1</code> <code>bne \$at, \$zero, L</code>

- `$at` (registro 1): assembler temporaneo

# Produrre un modulo oggetto

- L'assembler (o il compilatore) traduce un programma in istruzioni macchina
- Fornisce informazioni per costruire un intero programma da diversi pezzi:
  - Intestazione: descrizione dei contenuti del modulo oggetto
  - Segmento di testo: istruzioni tradotte
  - Segmenti dei dati statici: dati allocati per la durata del programma
  - Informazioni di rilocazione: per i contenuti che dipendono dalla locazione assoluta del programma caricato in memoria
  - Tabella dei simboli: definizioni globali e riferimenti esterni
  - Informazioni di debug: per associare il programma al codice sorgente

# Collegare un modulo oggetto

- Produce un'immagine eseguibile
  - Unisce i vari segmenti
  - Risolve le etichette (determina i loro indirizzi di memoria)
  - Corregge i riferimenti interni ed esterni
- Può lasciare dipendenze di locazione da sistemare tramite un loader rilocante
  - Ma con la memoria virtuale non ce n'è bisogno
  - Il programma può essere caricato in una locazione assoluta nello spazio della memoria virtuale

# Caricare un programma

- Caricare da un file immagine sul disco in memoria
  1. Leggere l'intestazione per determinare le dimensioni del segmento testo e del segmento dati
  2. Creare lo spazio di indirizzamento per contenere i segmenti
  3. Copiare istruzioni e dati in memoria
  4. Inizializzare gli argomenti sullo stack passati al programma principale
  5. Inizializzare i registri (inclusi `$sp`, `$fp` e `$gp`)
  6. Saltare alla procedura di avviamento
    - Copia gli argomenti `$a0`, ... e invoca `main`
    - Quando `main` termina, invoca la procedura di sistema `exit`



# Collegamento dinamico

- Collega/carica una procedura di libreria (.dll/.so) quando è invocata
- Richiede che il codice della procedura sia rilocabile
- Evita esplosione del codice a causa del collegamento statico di tutte le librerie indirizzate (transitivamente)
- Sceglie automaticamente le nuove versioni delle librerie

# La procedura scambia in C

- Illustriamo l'uso di istruzioni assembly per una funzione *bubblesort* in C
- Procedura scambia (foglia)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- $v$  in  $\$a0$ ,  $k$  in  $\$a1$ ,  $temp$  in  $\$t0$

# La procedura scambia

- Codice MIPS:

```
scambia: sll $t1, $a1, 2    # $t1 = k * 4
          add $t1, $a0, $t1 # $t1 = v + (k * 4)
                               # (indirizzo di v[k])
          lw $t0, 0($t1)    # $t0 (temp) = v[k]
          lw $t2, 4($t1)    # $t2 = v[k + 1]
          sw $t2, 0($t1)    # v[k] = $t2 (v[k + 1])
          sw $t0, 4($t1)    # v[k + 1] = $t0 (temp)
          jr $ra            # ritorno alla procedura chiamante
```

# La procedura ordina in C

- Non foglia (invoca scambia)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            scambia(v, j);
        }
    }
}
```

- v in \$a0, n in \$a1, i in \$s0, j in \$s1

# Il corpo della procedura

	move \$s2, \$a0	# salva \$a0 in \$s2
	move \$s3, \$a1	# salva \$a1 in \$s3
	move \$s0, \$zero	# i = 0
for1tst:	slt \$t0, \$s0, \$s3	# \$t0 = 0 se $\$s0 \geq \$s3$ ( $i \geq n$ )
	beq \$t0, \$zero, exit1	# vai a exit1 se $\$s0 \geq \$s3$ ( $i \geq n$ )
	addi \$s1, \$s0, -1	# j = i - 1
for2tst:	slti \$t0, \$s1, 0	# \$t0 = 1 se $\$s1 < 0$ ( $j < 0$ )
	bne \$t0, \$zero, exit2	# vai a exit2 se $\$s1 < 0$ ( $j < 0$ )
	sll \$t1, \$s1, 2	# \$t1 = j * 4
	add \$t2, \$s2, \$t1	# \$t2 = v + (j * 4)
	lw \$t3, 0(\$t2)	# \$t3 = v[j]
	lw \$t4, 4(\$t2)	# \$t4 = v[j + 1]
	slt \$t0, \$t4, \$t3	# \$t0 = 0 se $\$t4 \geq \$t3$
	beq \$t0, \$zero, exit2	# vai a exit2 se $\$t4 \geq \$t3$
	move \$a0, \$s2	# 1o parametro di scambia è v (vecchio \$a0)
	move \$a1, \$s1	# 2o parametro di scambia è j
	jal scambia	# invoca procedura scambia
	addi \$s1, \$s1, -1	# j -= 1
	j for2tst	# salta a test del ciclo interno
exit2:	addi \$s0, \$s0, 1	# i += 1
	j for1tst	# salta a test del ciclo esterno

# L'intera procedura

ordina:	addi \$sp,\$sp, -20	# fare spazio per 5 registri
	sw \$ra, 16(\$sp)	# salvare \$ra sullo stack
	sw \$s3,12(\$sp)	# salvare \$s3 sullo stack
	sw \$s2, 8(\$sp)	# salvare \$s2 sullo stack
	sw \$s1, 4(\$sp)	# salvare \$s1 sullo stack
	sw \$s0, 0(\$sp)	# salvare \$s0 sullo stack
	...	# corpo della procedura
	...	
exit1:	lw \$s0, 0(\$sp)	# copiare \$s0 dallo stack
	lw \$s1, 4(\$sp)	# copiare \$s1 dallo stack
	lw \$s2, 8(\$sp)	# copiare \$s2 dallo stack
	lw \$s3,12(\$sp)	# copiare \$s3 dallo stack
	lw \$ra,16(\$sp)	# copiare \$ra dallo stack
	addi \$sp,\$sp, 20	# copiare lo stack pointer
	jr \$ra	# ritornare alla procedura chiamante

# Intel x86 ISA

- Evoluzione con retro-compabilità
  - 8080 (1974): microprocessore a 8 bit
    - Accumulatore, più 3 coppie indice-registro
  - 8086 (1978): estensione a 16 bit del 8080
    - Insieme complesso di istruzioni (CISC)
  - 8087 (1980): coprocessore in virgola mobile
    - Aggiunge l'FPU e il registro stack
  - 80286 (1982): indirizzi a 24 bit, MMU
    - Memoria segmentata e protezione
  - 80386 (1985): estensione a 32 bit (ora IA-32)
    - Modi di indirizzamento e operazioni aggiuntionali
    - Memoria paginata e segmentata

# Intel x86 ISA

- Ulteriore evoluzione...
  - I486 (1989): pipelining, cache e FPU su chip
    - Avversari compatibili: AMD, Cyrix
  - Pentium (1993): superscalare, datapath a 64 bit
    - Le ultime versioni hanno aggiunto istruzioni MMX
    - L'infame bug FDIV
  - Pentium Pro (1995), Pentium II (1997)
    - Nuova microarchitettura
  - Pentium III (1999)
    - Aggiunte istruzioni SSE e registri associati
  - Pentium IV (2001)
    - Nuova microarchitettura
    - Aggiunte istruzioni SSE2



# Intel x86 ISA

- E ancora...
  - AMD64 (2003): architettura estesa a 64 bit
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adottata da Intel (con modifiche)
    - Aggiunte istruzioni SSE3
  - Intel Core (2006)
    - Aggiunte istruzioni SSE4, support per macchine virtuali
  - AMD64 (2007): istruzioni SSE5
    - Intel ha deciso di fare altro...
  - Advanced Vector Extension (2008)
    - Registri SSE più lunghi, più istruzioni
- Se Intel non avesse esteso la retro-compabilità, i suoi avversari lo avrebbe fatto!
  - Eleganza tecnica  $\neq$  successo di mercato

# Registri x86 di base

Nome	Utilizzo
31	0
EAX	GPR 0
ECX	GPR 1
EDX	GPR 2
EBX	GPR 3
ESP	GPR 4
EBP	GPR 5
ESI	GPR 6
EDI	GPR 7
CS	Puntatore al segmento di codice
SS	Puntatore al segmento di stack (cima dello stack)
DS	Puntatore 0 al segmento dati
ES	Puntatore 1 al segmento dati
FS	Puntatore 2 al segmento dati
GS	Puntatore 3 al segmento dati
EIP	Puntatore all'istruzione (PC)
EFLAGS	Condizioni

# Note conclusive

- Principi di progettazione
  1. La semplicità favorisce la regolarità
  2. Più piccolo è più veloce
  3. Rendere il caso comune veloce
  4. La buona progettazione richiede buoni compromessi
- Livelli di software/hardware
  - Compilatori, assembler, hardware
- MIPS: tipico esempio di ISA RISC
  - da confrontare con x86 (CISC)